UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

PEDRO SASSEN VEIGA

Light Programming Language

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Rodrigo Machado

Porto Alegre
December 2018

ACKNOWLEDGMENTS

We thank Jonathan Blow for the inspiration to create a complete language.

ABSTRACT

The increase of computing power in the last decades allowed for the creation and establishment of many high level programming languages such as Java and Python. In these languages, control over the hardware is often neglected in favor of more convenient abstractions for the programmer that offer some important guarantees (such as memory safety). At the same time, older lower level languages, such as *C*, are still considered one of the few viable options for systems programming. This work proposes a new low level programming language called *Light* that makes use of meta-programming ideas, commonly present in higher level, interpreted languages, in a compiled one. *Light* is a lower level, statically typed language that focuses on simplicity, consistent syntax and understandability. It has minimal runtime, no garbage collection and is composed of a simple core with a meta-programming layer built on top. We will present the complete language design and its compiler implementation. The objective of this work is to provide a general purpose system language that uses meta-programming to complement the base language as a tool to the programmer for building software.

Keywords: Linguagens de Programação. Meta-programação. Compiladores.

RESUMO

O aumento em poder computacional nas últimas decadas permitiram a criação e estabelecimento de diversas linguagens de programação de alto nível como Java e Python. Nessas linguagens, controle sobre o hardware é constantemente esquecido em favor de abstrações mais convenientes para o programador que oferencem algumas garantias importantes (como segurança de memória). Ao mesmo tempo, antigas linguagens de baixo nível como *C*, ainda são consideradas uma das poucas alternativas para linguagens de sistema. Esse trabalho propõe uma nova linguagem de programação de baixo nível chamada *Light* que faz uso de conceitos de meta-programação, comumente presentes em linguagens interpretadas de alto nível, em uma linguagem compilada. *Light* é uma linguagem de baixo nível, estaticamente tipada com foco em simplicidade, consistência de sintaxe e compreensibilidade. Possui ambiente de execução mínimo, não possui coletor de lixo e é composta de um núcleo simples com uma camada de meta-programação construída por cima. Nós apresentaremos o projeto completo da linguagem e a implementação de seu compilador. O objetivo deste trabalho é oferecer uma linguagem de sistema de uso geral que utiliza-se de meta-programação para complementar a linguagem base como uma ferramenta para o programador construir *software*.

Palavras-chave: Programming Language, Meta-Programming, Compilers.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

CTE	Compile time execution
CM	Code modification
RTTI	Runtime type information
SIMD	Single Instruction Multiple Data
ALU	Arithmetic and Logic Unit
AST	Abstract Syntax Tree
IR	Intermediate Representation
API	Application Programming Interface

CONTENTS

1 INTRODUCTION

1.1 Background

Simplicity is often overlooked in modern language design. With almost all new languages since the creation of the first programming languages, with a few exceptions, feature creep and patched features are common place nowadays. The programmer is almost always forced to work with several languages that have several thousand pages of documentation and are still changing. Since the 1950's, when the first programming languages were created, the evolution branched out into many different types of languages. But as for low level, "close to the metal" languages, few of them survived until today. Notoriously, the C programming language, proposed in 1972 by Dennis Ritchie and Ken Thompson (KERNIGHAN, 1988) is still to this day used for embedded systems, low level and systems programming. Inspired by Simula, an early object oriented programming language, in 1979 Bjarne Stroustrup developed C++ to be an evolution of C. Like Simula, C++ is an object oriented language but tries to take C's place in the low level language niche while also maintaining full backwards compatibility with C. Since then, an impressive amount of effort was made in the programming language field. However, the main focus was dedicated to higher level languages, leaving C and C++ almost by themselves as low level programming languages.

In the last couple of decades, a tremendous amount of effort was put into making higher level languages fit programmers needs in a way that removed them from the hardware beneath. It is not a surprise that this effort gave birth to many of the most popular languages today, like *Python*, *JavaScript*, *PHP*, *Java*, *C#* and many others. Almost all of them have very similar goals. Many of them were an attempt to simplify and automate web developing to be later adapted to general purpose use or vice-versa, gathering a substantial amount of features and libraries. Also along with many higher level languages, a few lower level focused languages emerged, like *D* (2007), *Rust* (2010) and even *Go* (2009), although the latter having other goals that will be later discussed. Along with these languages, the main inspiration for the *Light* programming language was Jonathan Blow's yet to be released *jai* (BLOW, 2014), which attempts to fill the niche of a low level, modern language just like *Light* does, but with a focus in games.

Interpreted languages by their nature, have the ability to execute code on the fly, as well as having a runtime type system that provides a lot of information to the programmer, making them very powerful and resourceful languages to work with. *Light* attempts to make those features available to low level programmers that understand their code in a more deep level, but maintaining a statically typed compiled language as a baseline. Reaching that goal brings myriad benefits with respect to quality of software, because a faster runtime program is always better for the end user. *Light* attempts to reach that goal using a simple language core that provides the feature set that is most important, possibly eliminating smaller supporting features in order to achieve less variability in the code.

1.2 Motivation

A programming language has as its primary goal to translate to a computer exactly what the programmer wants to do. For that, many approaches were taken and trade offs are unavoidable, so creating a perfectly expressive programming language for every different field and application might be impossible. Although creating abstractions to solve problems is a great way of doing things quickly, dealing with the hardware at a low level requires knowledge of many things like the architecture, system, memory layout and instruction set. All of those components have limitations, and in order to accomplish a more ambitious project, one would have to deal with those concepts. Assuming such task, transparency is imperative in the language - the abstractions a language has between the programmer and the hardware becomes just another mental construct to remember and keep in mind - transforming the programmer problem into a fight with the language in that case. Higher level constructs can be useful and are useful when they do not impose themselves when not needed.

Several modern languages provide large feature sets in order to speed up development. Language growth, although beneficial for few specialists in that specific language or technology, also comes with deleterious consequences like lack of coherence in syntax, unwanted or unused features, bad design decisions. It also creates a scenario where the same language can look and feel like other languages. Conversely, that are modern languages that prioritize simplicity, for example *Go*, which values simplicity, minimalism and coherence. It is not a systems level programming language, is garbage collected and still maintains a level of abstraction and similarity with object oriented languages. We will show in chapter 4 (Language comparisons) many examples that illustrate why the Light language was created and why its few features regarding meta-programming, code generation and code modification are important to modern low level languages. We believe that is still space for a language that facilitates systems programming whithout requiring a complex runtime support, relying on key features to accomplish better understandability.

Keeping a small and solid core language was paramount for the success of *C* and it is also the main influence for the design decisions that will be presented throughout this work. To give the programmer the tools required to write programs that still keep the hardware and performance as a concern is therefore a big motivation for this work and will manifest in design decisions and even limitations that will be explained in the following chapters.

1.3 Objectives

The main objective of the Light programming language is to be an alternative to *C* and *C++* as lower level languages for high performance, high bandwidth data processing, multi-threading and CPU intensive tasks. The language preserves a few core features from several languages whilst giving a solid and powerful meta-programming, code inspection, code modification and good support for code visualization and debugging. It is intended to be very pragmatic and loose - unrestrictive - not having security as a main priority, instead opting for a more pragmatic approach of being friendly to helping tools like debuggers and memory visualizers to provide compensation for that underrepresented area. This work will present the language state along with its initial compiler with an overview of the main features, design decisions, technologies used and a road map for future work. To minimize the difficulty of translation between a more human understandable language to a machine one is the goal of any programming language and the challenge is to do it in the most direct way possible.

2 LIGHT LANGUAGE

2.1 Overview

The Light language is based on a very simple core that underpins a meta-programming layer, which will be detailed in Section 2.3. This chapter will give an overview of the core language, its constructs and design decisions. As a statically typed compiled language, implementation will also appear as a major concern in design decisions since the language intent is to provide efficient runtime and fast compilation time. All language and compiler details will be abstracted in this chapter in order to present the language from the perspective of the programmer. Further details about the compiler and comparisons with existing languages will be later presented in subsequent chapters. Keeping the feature set to a minimum is also an objective, therefore all features that appear in the language were considered to be essential and sufficient to fulfill general programming needs. We recognize, however, that the perception that a reduced amount of features is advantageous can be highly subjective. Some of the main features that characterize the language are *type inference*, *compile time execution of code*, *code modification*, *reflection* and *introspection*.

2.2 Core

The Light programming language has a simple core that is the base for all other constructs. Having a simple core is important to reduce the amount of complexity when generating code to match what the programmer wrote. This avoids obtuse or seemingly strange behavior, from the perspective of the programmer, that is common in a more complex language like *C++*.

The core language is composed of three main kinds of constructions: declarations, expressions and commands. Compiler directives, for example *#run*, are excluded from the core and will be addressed subsequently in Section 2.3. A declaration will always have a name associated with it and can either be constant, identified by the token `::`, or not, identified by a single `:`. All declarations in the top level of compilation (global scope and file scope) are processed independent of order, without the need for header files or forward declarations. In the example code shown in Listing 2.1 the procedure *sum* is declared after *main* but is accessible by it independent of order.

Listing 2.1 – Example declaration order of top level

```
main :: () -> s32 {
        return sum(2, 3);
}

sum :: (a : s32, b : s32) -> s32 {
        return a + b;
}
```

All declarations inside a scope that is more internal than a file scope are dependent of order and will cause a compiler time *undeclared identifier error* in the event of using an identifier before its declaration. Declarations can be one of the following:

- Procedure
- Variable
- Constant
- Structure
- Enumeration
- Union
- Type Alias

Another construct of the language is the command, which directly dictates control flow and assignments. Most commands are control flow statements, with the exception of assignments and block delimiters. List of possible commands:

- Block
- Assignment
- If
- For
- While
- Break
- Continue
- Return

Finally, expressions allow one to express data types and operations over them. All arithmetic expressions, literals, memory manipulation and procedure calls are expressions. Unlike the *C* language, *Light* is more restrictive in relation to expressions. For example, the *C* ternary operation `(condition)? true_result : false_result`, which is the equivalent of a conditional if-then-else for expression, does not have an associated construction in *Light*. *Light* does not allow several of these constructs common to other languages in order to be clear and offer the minimum amount of features needed to accomplish the same goal. The list of possible expressions is:

- Binary expression
- Unary expression
- Literal
- Variable
- Procedure call
- Directive

2.2.1 Syntax

The first important part of a language is syntax. The focus of the Light language is to have consistent and orthogonal syntax. We intend for consistency to have priority over other design aspects such as beauty and conciseness. Having a simple and consistent foundation allows the programmer to reduce friction with the language constructs. By minimizing syntax variability, *Light* reduces the programmer need to remember the language's syntax, therefore improving productivity. This section will describe the "Light" syntax as it is at the time of this work. In Figure 2.1 we present all the reserved keywords of the Light programming language.

Figure 2.1: Light Keywords

bool	s16	if	return	true
void	s8	else	struct	false
r32	u64	for	enum	string
r64	u32	while	union	
s64	u16	break	array	
s32	u8	continue	null	

There are two types of comments in *Light*, the single line comment is characterized by double forward slashes, which comments everything after the slashes up until the end of line. There are also multi line comments, which start with the token `/*` and end with the token `*/`, commenting everything within those tokens. Multi line comments can also be nested.

Declarations always bind to a name and a type separated by a colon. For instance, the declaration `x : u32;` declares a variable `x` of type unsigned integer of 32 bits with default value of zero. Optionally, an assignment can immediately follow a declaration. For instance, the declaration `x : u32 = 3;` declares the same variable `x` and assign the value 3 to it. When accompanied by an assignment, the type can be optionally omitted, making use of type inference, which in the previous example would become `x := 3;`. This would change the type of `x` to be the default type for the literal 3 (`s64`).

Constant declarations are similar, only instead of an assignment, they are indicated by an extra colon (`:`). For instance, the declaration `main :: ()-> s32 { ... }` declares a procedure `main`, which returns a signed integer of 32 bits. In the case of other types (not functional), the type is declared between the colons. The code to declare a constant value `x` of type `u32` would be `x : u32 : 3;`.

In the code presented in the Liting 2.2, we declare a procedure `main` (line 1), a constant *MAX* (line 2) and a variable *sum* (line 3). The example also shows a for loop in the line 5, for a programmer of procedural languages with syntax similar to C's, the *Light* syntax for commands is very familiar, that is a design decision that will manifest also in the language semantics and has the intent of facilitate the transition from those languages to *Light*. The complete language grammar is found in the Appendix A.

Listing 2.2 – Light Syntax example

```
1 main :: () -> s32 {
2     MAX :: 10;
3     sum : s64;
4 
5     for i := 0; i < MAX; i += 1 {
6         sum += i;
7     }
8     return [s32]sum;
9 }
```

2.2.1.1 Type Declaration

The type declaration syntax is read left to right where the symbol ˆ (caret) is read *pointer to*. The array type is represented by brackets $[S]$, where S is the array size expression and is read *array of S*. The functional type starts with begin parenthesis `(` followed by a list of argument types, ending with a close parenthesis `)` and an arrow token `->`

followed by the procedure return value. The structure and union types are represented by its names, since there are declarations binding them to their respective definitions. Finally the primitive types are represented by the reserved keywords in the Table 2.1.

Table 2.1: Primitive type keywords

`u8 u16 u32 u64`	unsigned integers
`s8 s16 s32 s64`	signed integers
`r32 r64`	floating point numbers
`bool`	boolean type
`void`	unit (no value)

Using those rules, all types in the language can be built. The Table 2.2 shows examples of various type declarations in *Light* and its correspondent descriptions in natural language.

Table 2.2: Type declaration syntax examples

`[32]u8`	array of 32 u8's
`^[4]bool`	pointer to array of 4 booleans
`() -> ^s32`	procedure with no arguments returning pointer to s32
`((s32, s32)-> s32)-> void`	procedure receiving a procedure receiving two s32's and returning s32 and returning void
`[10]()->()->s32`	array of 10 procedures with no arguments returning a procedure with no arguments and returning s32

2.2.1.2 Literals

Literals are the values of types that can be directly expressed in the source code of the language, such as numbers, string or structures. Currently, *Light* does not provide a literal representation for functions (lambda notation), although we can declare functions, create variables that store them, assign functions as values and pass them as arguments to higher-order functions. Union literals are also not present in the language.

The most simple type of literals are integer and floating point that represent integers and floating point types respectively. The rules for the lexical tokens are described using regular expressions in the Appendix A along with the language grammar. Integer literals can be expressed in decimal, hexadecimal and binary while floating point currently only support the standard syntax without scientific notation. Other primitive type literals are booleans, represented by the reserved keywords `true` and `false`. The `void` type does not have a literal representation. Character literals are syntactic sugar for unsigned 32 bit integers that are translated to the character's Unicode representation.

Pointer types are an exception for literal construction since the only pointer value represented by a literal is the *null pointer* value, which is represented by the reserved keyword `null`. Other values for pointer types can only be extracted using operations. For instance the code `&x` where `x` is an addressable value of type `T`, represents the pointer value to a value of type `T`.

Arrays and structures (records) are non-atomic structures which support arbitrary nesting. Because of this, it is important to follow the principle of a clear syntax, that is, maintaining a construction pattern the simplest possible. Array literals therefore are constructed recursively following the pattern `array:{L1, L2, ..., Ln}`, where `L1` is a literal of the type of the array separated by colons inside brackets. Given that literals are finite, the element count will determine the array dimension.

Similar to the array literal, the structure literal is a recursive construction following the `StructName:{L1, L2, ..., Ln}` pattern. As expected, the order and types of literals `L1, L2, ..., Ln` must abide the format established by the struct declaration. The string type in the *Light* language is implemented as a syntactic sugar for a internally defined struct declaration (2.3). As an example, the string literal `"Hello World!"` is syntactic sugar for `string:{12, -1, &arr}` where `arr` is an array of characters `arr : [12]u8 = {'H', 'e', ...}`.

Listing 2.3 – Light string declaration

```
string :: struct {
    length   : s64;
    capacity : s64;
    data     : ^u8;
}
```

In the example Listing 2.4 a literal for a structure `Vertex` is nested with an array literal, making an array of four `Vertex`. In line 10 a pointer is declared and initialized using the `null` literal. Line 13 shows an example of a boolean variable declaration and line 16 shows a `string` declaration using a literal.

Listing 2.4 – Light literals example

```
1  // Array, struct and floating point literals.
2  vertices : [4]Vertex = array {
3     Vertex:{vec3:{-1.0, -1.0, 1.0}, vec2:{1.0, 1.0}},
4     Vertex:{vec3:{ 1.0, -1.0, 1.0}, vec2:{1.0, 1.0}},
5     Vertex:{vec3:{ 1.0,  1.0, 1.0}, vec2:{1.0, 1.0}},
6     Vertex:{vec3:{-1.0,  1.0, 1.0}, vec2:{1.0, 1.0}},
7  };
8
9  // pointer literal
10 ptr : ^s32 = null;
11
12 // boolean literal
13 boolean := true;
14
15 // string literal
16 name := "Literals example";
```

2.2.1.3 Commands

An assignment in the *Light* language is a command that operates on two expressions, much like a binary expression, although a command does not have a return value and cannot be used inside an expression. The left side of an assignment is called the *lvalue* and the right side, *rvalue*, in many languages and in *Light* likewise. An assignment operation is represented by the token `=` with many syntactic sugar variations of the binary operations. `+=` | `-=` | `*=` | `/=` | `%=` | `<<=` | `>>=` | `^=` | `&=` | `|=`
All of which are syntactic sugar for `lvalue = lvalue BINARY_OPERATION rvalue`, i.e. `a += b` is equivalent to `a = a + b`.

Control flow commands in the *Light* language are for the most part composed by a starting keyword followed by expressions or more commands. The standard branching command is the `if` statement. In *Light*, differently from *C/C++*, the *if* keyword is followed immediately by a boolean expression, very similar to *Go's* syntax. Like in most languages, an `else` statement can occur optionally after an *if*. In the example 2.5 the first two *if* statements are equivalent and the last (line 9) doesn't make use of an *else*.

Listing 2.5 – Light if/else example

```
1 if a >= b {
2    return a + b;
3 } else {
4    return a - b;
5 }
6
7 if a >= b return a + b; else return a - b;
8
9 if a >= b return -1;
```

Even simpler than the `if` statement, the `while` command does not have an optional `else`, hence will always follow the pattern $while\ expression\ command$. An example of an infinite loop would be: `while true {}` since an empty scope block is a command.

As in most languages, the `while` command is complemented by other looping constructs with an objective of convenience and conciseness. These constructs are present in the *Light* language in the form of syntactic sugar. In the current version of the language the `for` statement is the only construct built over the `while` command. The structure is similar to the one used in *C*, starting with initializers commands separated by commas followed by a semicolon, an exit condition expression, semicolon, posterior loop commands and finally the command to run inside the loop. The example 2.6 illustrates the use of the `for` command to calculate the sum of the numbers between 0 and 10 with its respective syntax expansion and output.

Listing 2.6 – Light for loop example

```
for i := 0, sum := 0; i < 10; i += 1 {
    sum += i;
    print("% ", sum);
}

// Expands to:
{
    i := 0;
    while i < 10 {
        sum = sum + i;
        print("% ", sum);
        i += 1;
    }
}
```

```
$ 0 1 3 6 10 15 21 28 36 45
```

Complementing the control flow statements are the commands `break`, `continue` and `return`. All of which can appear by themselves or followed by an expression, which in the case of the `return` command corresponds to the return value of the scoping procedure. The other constructs can only appear inside a loop and optionally followed by an integer literal, which will be later explained in the *type system* section 2.2.2.

Finally, a command block serves two purposes, aggregating a sequence of commands as a single command and providing an explicit scope for internal declarations. The command block body can be viewed as a list of commands and declarations separated by semicolons and delimited by curly brackets. An empty block is also considered valid in *Light*.

2.2.1.4 Expressions

An expression is build from literals, variables and operations. Additionally, it can also be built from directives, as seen in Section 2.3. An important distinction from the

C/C++ language is the fact that an assignment is not an expression in *Light*. Consequently, *C* expressions such as `i++` are not allowed in *Light*. This is to avoid common syntax misinterpretations and keep the intent of an expression more clear to the reader.

Binary operators comprise the largest number of operators available in *Light*. The most common processor level operations like addition, subtraction, bit shifting and comparisons are binary operations taking two expressions in the form `expr op expr` where `op` is one of the operators listed in the table 2.3 Binary operators. The only exception to the general rule is the array accessing operator which is in the form `expr[expr index]`.

Table 2.3: Binary operators

`+ - * / %`	arithmetic operators
`<< >> ^ & |`	bitwise operators
`&& ||`	logic operators
`< > <= >= == !=`	comparison operators
`.`	dot operator
`[]`	array accessing operator

Along with binary operators, the unary operators in the *Light* language also characterize some of the most common operations found in a processor ALU. As it is intended to be used for systems programming, memory operations like *address of* and pointer *dereference*, as well as *type casting* operations are necessary for the intent of the language and therefore are present. All unary operators are prefixed in the expression in the form `unop expr` and are also inspired in the *C* language syntax, being very similar with the exception of the casting operator which uses brackets instead of parenthesis. The table 2.4 shows all unary operators available in the language and its description.

Table 2.4: Unary prefixed operators

`+`	unary plus
`-`	unary minus
`~`	bitwise not
`!`	logic not
`&`	address of
`*`	dereference
`[Type]`	unary cast

Operator precedence differs between languages and is common cause of confusion, therefore must be designed with care in order to avoid surprises. The *Light* language follows an operator precedence table 2.5 with a left to right associativity for binary operators and right to left associativity for unary operators. The precedence table 2.5 has precedence in ascending order from top to bottom. As an example, a binary expression `a + b + c` is equivalent to `((a + b)+ c)` while the expression `-*v` will be equivalent to `-(*(v))` and the expression `a + b * c` according to the table is equivalent to `(a + (b * c)` since multiplication has higher precedence than addition.

Table 2.5: Operator precedence table

`&&` `||`
`<` `>` `<=` `>=` `==` `!=`
`<<` `>>` `^` `&` `|`
`+` `-`
`*` `/` `%`
unary operators
`.`
`[]` *procedure call*
parenthesis

2.2.2 Type System

A statically typed language, as *Light*, associates a specific type to well-formed programs, and such types are intended to be preserved by program evaluation. Contrary to languages like *C++*, that are statically typed but incorporate some parts of runtime type evaluation in a form of object oriented polymorphism, the *Light* language is completely static. Since one important objective is to maximize performance while keeping a modern, simple and easy to use language, *Light* provides type inference optionally at variable or constant declarations, meaning it will infer the type based on the *rvalue* in the initialization assignment of the declaration. This makes the syntax more concise and also provides convenience for the programmer that does not need to explicitly declare the type of all declarations. The Listing 2.7 shows an example of type inference, where declarations in lines 8, 9 and 10 are inferred from their respective initial assignments by omitting the type declaration after the colon.

Listing 2.7 – Light type inference example

```
vec2 :: struct {
    x : r32;
    y : r32;
}

main :: () -> s32 {
    normal : s32 = 1;   // normal declaration with the type.
    one    := 1.0;      // r32 inferred.
    vector := vec2:{2.0, 3.0};  // inferred as vec2
    arr    := array {1, 2, 3};  // inferred as [3]s64

    return 0;
}
```

Light also defines types as being unique and available as values in run-time, allowing programming techniques that depend on availability of type information at runtime, such as reflection, for instance. This is known in many languages as runtime type information (RTTI).

2.2.2.1 Operations

The *Light* type system is quite restrictive regarding implicit type coercions, keeping them at the minimum. At the current language state there exists only one type coercion which converts any pointer type to `void`. Many languages choose to keep a big type coercion table to allow programmers to write more freely without worrying about type errors, often ignoring unsafe type coercions warnings. The policy for *Light's* type system is to not give any warnings, therefore currently every type mismatch will raise a type error.

To describe the type system in depth we will use the Table 2.6. In the semantic rules used to describe the *Light* type system, the left side of the symbol $\mapsto$ represents an operation using the types specified and the right side the resulting type from the operation.

Table 2.6: Types and Operators

Types	*Description*
`u8 u16 u32 u64`	Integer unsigned
`s8 s16 s32 s64`	Integer signed
Integer unsigned *Integer signed*	Integer
`r32 r64`	Floating point
`bool`	Boolean
Operators	
`+ - * /`	Arithmetic
`%`	Modulo
`< > <= >= == !=`	Comparison
`<< >> | & ^`	Bitwise
`&& ||` *Comparison*	Boolean

Binary operations in *Light* are well defined and do not allow for coercion of any type in the current state of the compiler. An incorrect typed construction will cause a type mismatch error which indicates that a valid operation is done with incompatible types. Bypassing this can be done with type casting, explained in more detail at the end of this chapter. Overflow and underflow, as well as representation limits are present in the language but we omit them for the sake of brevity. Unsigned integers obey arithmetic modulo rules according to the number of bits in its representation. Floating point values and operations follow the IEEE 754 standard (IEEE..., 2008) (Same as Intel's modern chips).

All valid binary operations of primitive types and its corresponding type yields are described in the rules below, where lines with types of the same description are equal, i.e. $Integer + Integer \mapsto Integer$ where Integer is *u32* means $u32 + u32 \mapsto u32$.

Figure 2.2: Binary operations - Type rules

$$
\begin{array}{llllll}
Integer & Arithmetic & Integer & \mapsto & Integer \\
Integer & Bitwise & Integer & \mapsto & Integer \\
Integer & Comparison & Integer & \mapsto & bool \\
Floating\ point & Arithmetic & Floatint\ point & \mapsto & Floating\ point \\
bool & Boolean & bool & \mapsto & bool \\
bool & == & bool & \mapsto & bool \\
bool & != & bool & \mapsto & bool \\
bool & \wedge & bool & \mapsto & bool
\end{array}
$$

All unary operations available are described in the rules below, where T is any type.

Figure 2.3: Unary operations - Type rules

$$
\begin{array}{lll}
- & Integer & \mapsto Integer \\
+ & Integer & \mapsto Integer \\
- & Floating\ point & \mapsto Floating\ point \\
+ & Floating\ point & \mapsto Floating\ point \\
\sim & Integer & \mapsto Integer \\
! & bool & \mapsto bool \\
* & \hat{}T & \mapsto T \\
\& & T & \mapsto \hat{}T
\end{array}
$$

Pointer arithmetic is an important construct for memory manipulation. Similar to *C*, the semantic of a sum and subtraction by an *integer type* is to multiply the integer with the size of the type pointed to. Because memory manipulation is an important concept for this language, safety of the type system is not guaranteed, since free manipulation of memory does not always guarantee a valid pointer will be return by pointer arithmetic operations.

Listing 2.8 – Light pointer arithmetic

```
a : ^s32 = [^s32]array{1, 2, 3};
b : ^s32 = a + 2; // a + (2 * #sizeof s32)
c : s32 = *b;      // c will have 3
```

Considering T a pointer of any type except `void`, the following semantic rules represent pointer arithmetic in the *Light* language:

$$
\begin{array}{llll}
\hat{}T & + & integer\ type & \mapsto \hat{}T \\
\hat{}T & - & integer\ type & \mapsto \hat{}T \\
\hat{}T & - & \hat{}T & \mapsto s64 \\
\hat{}T & comparison & \hat{}T & \mapsto bool
\end{array}
$$

With the aim to provide memory manipulation capabilities, type punning and compatibility with low level calling conventions, *Light* provides an *unary cast* operator. The Listing example 2.9 shows a reinterpretation of the memory for the value 3 as an `r32` floating point value using unions and unary operations.

Listing 2.9 – Light type punning

```
value :: union {
   f : r32;
   i : s32;
}

main :: () -> s32 {
   number : value;
   number.i = 3;
   reinterpreted_as_r32 : r32 = number.f;
   reinterpreted_as_u32 : u32 = *[^u32]&number.f;
}
```

Compatibility with *C's* standard calling convention was also decisive in choosing to keep this unsafe behavior. Unions in *Light* are untagged in order to preserve compatibility. Numeric types can be casted to any other numeric type (Floating point and integers). All other type casts are described by the following rules, considering T and S *any* types, different or not.

$$\begin{array}{cccc} \texttt{[\^{}}T\texttt{]} & Integer & \mapsto & \texttt{\^{}}T \\ \texttt{[\^{}}T\texttt{]} & \texttt{\^{}}S & \mapsto & \texttt{\^{}}T \\ \texttt{[\^{}}T\texttt{]} & Array\ Type & \mapsto & \texttt{\^{}}T \\ \texttt{[\^{}}T\texttt{]} & Functional\ type & \mapsto & \texttt{\^{}}T \end{array}$$

Currently enumerations are internally implemented by means of integer types, defaulting to `u32`. Therefore all previous rules regarding integers are applied to enumerated values in the language.

2.2.3 Commands

All commands with conditional operations (if, while and for) require a boolean typed expression. The *return* command is matched with the return type of the function that scopes it, meaning a type mismatch error is raised in case of conflicting types, similar to a binary operation.

The commands break and continue are required to be inside a looping command (while, for) and may optionally be followed by an integer literal which represents the level number from which the command is to break or continue.

Listing 2.10 – Light type punning

```
for i := 0; i < 10; i += 1 {
    for j := 0; j < 10; j += 1 {
        print("%", j);
        if(i == 3 && j == 5)
            break 2;
    }
    print("\n");
}
```

```
0123456789
0123456789
0123456789
012345
```

The loop depth level for a break or continue starts at 1. The level is checked and should not pass the number of nested loops or an error will be raised. The example 2.10 shows two levels of iterative loop with a break of two levels, this with result in breaking outside both loops when the condition inside the if statement is met, the output shows the result of the inside print statement.

Similar to a binary operation, an assignment command will cause a type mismatch if the types associated with the *lvalue* and *rvalue* do not match. Coercions are applied to assignments and transform the *rvalue* expression into the *lvalue* type before the assignment. This is also valid for assignments in declarations.

2.3 Meta-Programming

As languages evolve, the software industry has increasing demand for code, meaning code generation and the ability to inspect code should grow concurrently. This is not the reality since most statically typed languages give very limited or even no meta-programming ability. This concept is not new for interpreted languages, where running code generated ”on the fly” was never a problem, since the interpreter is doing this anyway at every line of code. The real challenge is to build a compiled statically typed language with a simple and reliable alternative that can mimic such feature in a simple and helpful way. This would be very beneficial not only for code optimization, but also automization of repetitive tasks and even customizable compile time code checking, improving code quality in general. This chapter shows important key mechanisms to achieve this goal. We intend to use the same language to write programs and to do meta-programming. Reducing variability in the language aims at a simple and understandable meta-programming capability. Although only compile time code execution was implemented for the initial version of the compiler. All those different features will be refered as a meta-programming layer on top of the core language which was already presented.

2.3.1 Compiler directives

Compiler directives represent the meta-programming layer. This layer is characterized by directives, which are:

- #run
- #import
- #if
- #else
- #assert
- #export
- #sizeof
- #typeof
- #compile
- #end
- #foreign

The directive `#sizeof` will take a type and return its size in bytes, since type sizes are known at compile time, this directive will generate an integer literal in place of the directive. A common use for this directive is dynamic allocation depending on the type of a structure or array to perform copies or simply comunicate with external API's.

Having type information at runtime provides capabilities for manipulating types as if they were values, sometimes refered to as reflection. The directive `#typeof` takes an expression and inserts in place of the directive a structure that represents the type and can be used at runtime. In the subsection 3.4.1 (Type Table) this is further explained along with the reference to the definition for the types in code.

Importing files is temporarily in the language only to organize projects into different files, since the current behavior is to add the imported file to the project as if it was pasted in the main file. As we intend to provide a proper module system in the future, where library imports and modules will be added, this is directive is planned to be altered.

Perhaps one of the most important directives, the `#foreign` directive follows a procedure declaration that will not have a body since it is an external imported procedure. This is directly compatible with *C* libraries and is designed to be simple to use. The library name comes after the `#foreign` directive like in the example

```
malloc :: (size : u64)-> ^void #foreign("c");
```

which imports from the *C standard library* the memory allocation procedure `malloc`.

2.3.2 Compile time code execution

Composed by the directives `#run`, `#assert` and `#if/#else`, compile time execution of arbitrary code can be used to generate constant expressions at compile time without restrictions. A simple example would be a table of hashes for keywords generated using the `#run` directive. In the code shown in Listing 2.11, a hash function generates the hash at compile time for the keywords `if` and `else` a common operation in a compiler implementation.

Listing 2.11 – #run example

```
hash :: (in : string) -> u64 {
    // some hash function...
}

hash_table := array { #run hash("if"), #run hash("else") };
```

The conditional directives `#if` `#else` will `#run` the expression directly following the directive and will conditionally include in the compilation the source code immediately after the expression until it reaches the `#end` directive. In the example 2.12, a common way to write multiplatform programs is to check for a definition at compile time indicating the operating system the compiler is running.

Listing 2.12 – #if example

```
#if PLATFORM_WINDOWS
#import "windows.li"
#else
#import "linux.li"
#end
```

Similarly, the `#assert` directive is a static assertion that uses `#run` in a boolean expression, aborting compilation with an error in case the expression evaluates to `false`.

2.3.3 Code Generation

Code generation is one of the most powerful features in *Light*. The directive `#compile` takes a string parameter followed by a command block. The string is defined inside this block and can be modified to contain arbitrary code, including other `#compile` directives, the directive will include the argument string in the compilation. The code depicted in Listing 2.13 defines a procedure that takes many different types of arguments, which in many languages is called generic programming. Also to illustrate nested `#compile` directives, the arguments for the `sprint` procedure are duplicated 4 times using another defined procedure that takes a string and replicates it separating by commas.

Listing 2.13 – #compile example

```
#compile result {
        types := array { "s32", "u32", "s64", "u64" };

        for i := 0; i < #sizeof types / #sizeof string; i += 1 {
                sprint(result, "sum_% :: (a : %, b : %) -> % { a + b }"
                    , #compile repeat_string(types[i], 4));
        }
}
repeat_string :: (s : string, count : s64) -> string {
        result : string;
        for i := 0; i < c; i += 1 {
        if i != 0 sprint(result, ", ");
                sprint(result, "%", s);
        }
        return result;
}
```

2.3.4 Code modification

Directly accessing the program's Abstract Syntax Tree enables powerful custom tools for code analysis or modification. One could check, for example, if a particular global variable is assigned at any point during execution, not just statically but also at runtime. This could be achieved by inserting checking code at every assignment which includes the address of this particular variable, this ensures the code is correct considering what the programmer defines as correct customly. Although this feature is not currently in the language, there are several different approaches to define it. For example, one would be a messaging system attached to the compilation stage where the programmer could intercept the compiler when certain compilation events happened, all information currently availabe to the compiler could be made available to the programmer. This is similar to what Jonathan Blow defines in his language *jai* (BLOW, 2014). Code modification in this way is arguably complex and requires deep understanding of the language AST, but the benefits outwheigh the potential addition in complexity in this case. As the language's AST was also designed to be fairly small and simple, the addition of this feature can be considered justifiable.

3 COMPILER IMPLEMENTATION

Along with the language design, a compiler was developed for Windows and Linux operating systems and its source code is available in the link <https://github.com/Hoshoyo/Light>. The compiler is written in *C++* and *Assembly* without use of third party libraries, with the exception of the C runtime library. The compiler base architecture is similar to the architecture described in the section 1.2 of (AHO; SETHI; ULLMAN, 1986), consisting of lexical analysis, syntax analysis, semantic analysis, intermediate code generation, symbol table management and code generation. To implement meta-programming, the compiler runs multiple passes, evaluating directives each time it runs. It is important to mention that any code can be run at compile time with the use of a directive such as `#run`, without any restriction, potentially leading to infinite loops during compilation. That is a design choice, since giving the programmer the most amount of freedom is one of the *Light's* design principles. Many of the design decisions behind the compiler implementation are inspired by Jonathan Blow's language *jai* (BLOW, 2014) which by the time of this work is not yet released. Described in this chapter, the lexer, parser and type checker will be referred as the front end whilst the intermediate code generator and code generator will be referred as back end.

Figure 3.1: Compiler Architecture

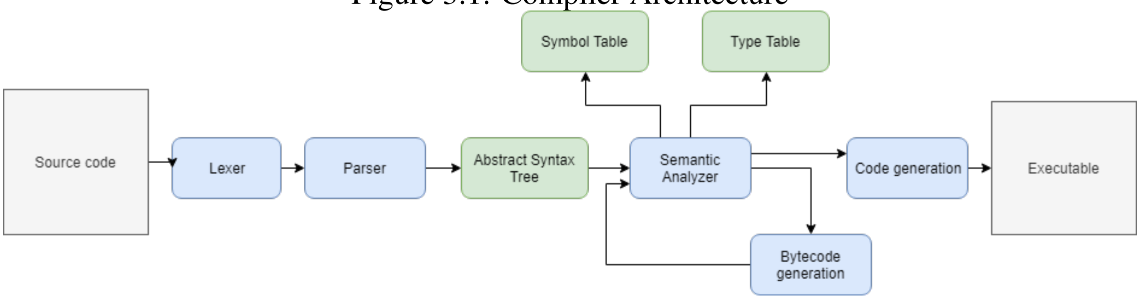


Figure 3.1 shows an overview of the compiler architecture. From the source code, the Lexer is invoked to provide tokens to the parser that, following the language grammar, available in Appendix A, constructs an Abstract Syntax Tree that is fed to the semantic analyzer which fills it with type information while creating symbol tables and one type table. Nodes that require another compilation step go through byte code generation and return to the semantic analyzer as part of the AST. At the end of type checking, code generation is performed to ultimately produce an executable.

3.1 Lexer and Parser

Reading all files and dividing them into tokens is the first step of the compiler, given initial files with *Light* source code. The lexer - also referred as tokenizer - is therefore responsible to keep important file information for describing the location of eventual syntax or type errors, as well as token type information and lexical range (token start in the stream along with its size). The lexer also is responsible for internalizing strings, that is, making a string unique in the compilation by using a hash table to facilitate later insertion in the symbol table. In that process it also identifies keywords and directive words, marking them accordingly. Comments and white spaces are ignores at this stage, differently from some languages like *Python*, which interprets indentations as being semantic meaningful.

With all lexical information the compiler proceeds to the parsing stage. This stage has as input the previous stage's data and as output a data structure representing the program AST. The parser uses a technique known as top-down recursive descent parsing, which is most natural for human understanding as opposed to a bottom up parser generated by a parser generating tool like flex/yacc (JOHNSON et al., 1975). A guide to implement a top down parser manually, similar to the one in this work can be found in the section 3.3 of the Modern Compiler Design book (GRUNE et al., 2012). It is in the parsing stage that syntax errors can be raised. These errors are caused by unexpected or missing tokens and are fatal to the compiler, halting compilation immediately, otherwise the compiler would have to guess the user's syntax mistake and might generate misleading errors.

Listing 3.1 shows the declaration of three variables in which the first two are declared without the ending semicolon. When the compiler is parsing the line 1, at the last token `0`, it expects either the end of the expression or the continuation of it, which could have been a binary operator or even an unary postfixed operator (currently non existent in Light). In the output it is clear that the compiler stopped at the first syntax error, at line 2 an unexpected token space was read.

Listing 3.1 – Syntax error example

```
i := 0
space := ' '
number := 2.0;
```

```
file.li:2:1 Syntax Error: expected ';', but got 'space'
```

3.2 The Abstract Syntax Tree

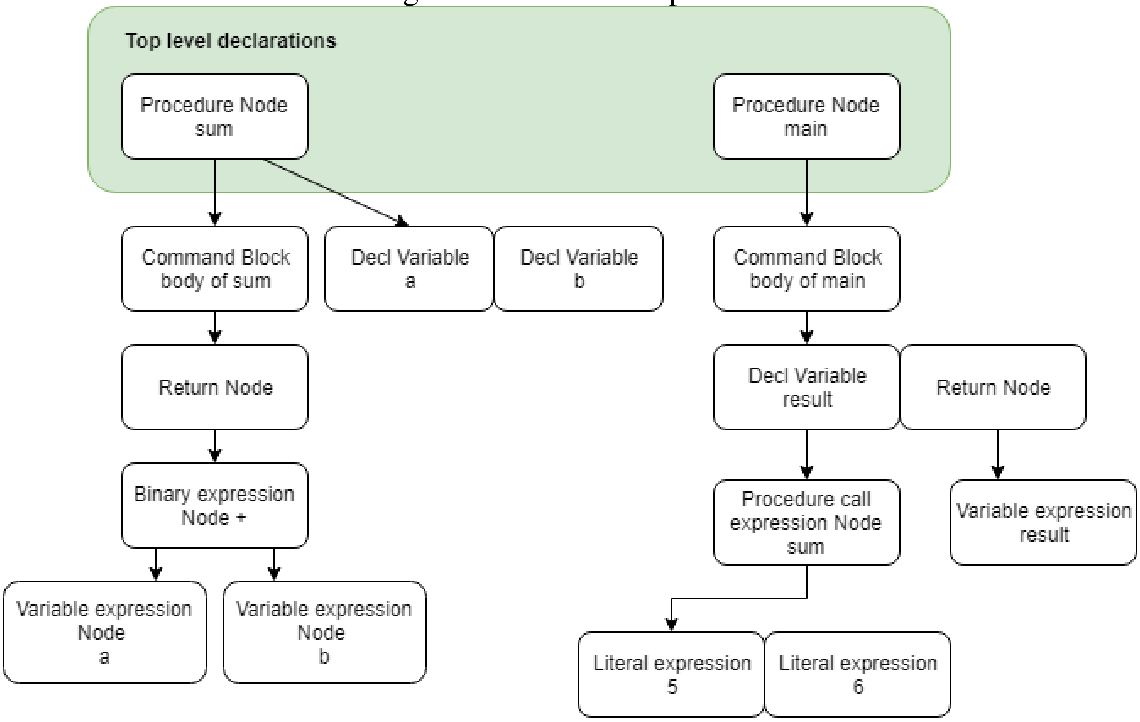
The abstract syntax tree is a representation of the program source code as a tree data structure containing all the required information for the semantic analysis and code generation steps. The information, if not provided directly by the parser stage, it is inferred in semantic analysis. Although all information is present in the AST, only simple language constructs - described in the Core language section 2.2 - will be part of it, meaning all syntactic sugar is processed in the parser stage. Each item in the lists from the Core language section is a node in the tree, which the definition can be found in the Appendix B directly transcribed from the original code.

The Figure 3.2 illustrates how the AST for the code in Listing 3.2 would be, omitting detailed information for clarity. Touching nodes in the diagram represent arrays of nodes of the same type, procedure arguments for the sum procedure and the commands inside the *main* procedure.

Listing 3.2 – Ast example code

```
sum :: (a : s32, b : s32) -> s32 {
    return a + b;
}
main :: () -> s32 {
    result := sum(5, 6);
    return result;
}
```

Figure 3.2: AST Example tree



3.3 Symbol Table and Scope

One of the major compiler operations is to perform identifier lookup, either to check for redeclarations or to retrieve information about an identifier. The information associated with each identifier, such as its type and size, is essential for routines such as type checking and code generation. To make this operation efficient, most compilers make use of a a hash table algorithm to optimize identifier lookup. Since good hash table look up implementations have constant ($O(1)$) asymptotic cost (Section 1.2.11 of (KNUTH, 1997) for the O notation), it is an efficient method that we employed in our implementation.

Each command block is part of a tree of symbol tables that define a scope, the top level global scope being the root branching down for every procedure block and nested blocks inside it. An identifier is considered defined if it is in any of the parent scopes to the one that it is used in or in the latter, if that path to the root defined the identifier more than once, the closest block to the one the identifier is used will shadow all the others and will be the valid declaration in that case. Redefinition errors also benefit from a fast identifier look up since they can refer to previous definitions and get their information to better describe what is the cause of a given error. For example, in Listing 3.3 we show two variable declarations (line 1 and 2) defined in global scope. In line 4, the variable max is defined again, shadowing the previous declaration, since its own scope definition precedes all previous ones. The definition of the variable avg (line 6) is confined to its scope only, since that are no previous definitions

Listing 3.3 – Scope rules example

```
1 max : s32 = 10;
2 min : s32 = -10;
3 main :: () -> s32 {
4     max := 255;
5     {
6         avg : s32 = 0;
7     }
8 }
```

The *Light* compiler makes use of the previous lexer work of internalizing strings to speed this process even more, utilizing its address in memory (as it is unique for each identifier) as a hash, making the comparison a simple register size comparison for any processor assuming the address size matches the register size.

3.4 Type Inference and Type checking

Semantic analysis is the last step where the AST is filled with information that will be used for the code generation, which comprises any back end for any architecture or even an intermediary language like LLVM's IR (LLVM. . . ,). At this stage, the compiler goes through the AST, inferring type annotations the programmer omited and perform type checking. Definitions without a type declaration will have their type assigned to the same type inferred in the expression inference step, this will ensure that all definitions will have a type associated with them. Structure and union type declarations are also internalized and considered a *strong* type by default. During this step the memory sizes and alignments are calculated for each field, and although not finished, memory alignment rules are by default the size of the type with byte padding (equal to *C's* default alignment rules). We plan to add alignment directives to the compiler in the future.

The type inference algorithm implemented uses the concept of *weak* and *strong* types. *Strong* types will force *weak* types to coerce to them, meaning for example, a *weak* s64 will coerce to any integer type since no type was specified. A *strong* type in contrast will force any other type to try to coerce to it. For example, a variable expression is always considered *strong* and therefore in the expression `variable + 10` where `variable` is of type `s16`, the literal `10` will coerce to an `s16` type. The default values for numeric literals are `s64` for integers and `r32` for floating point values, the reason for this choice is to match current technology since currently most processors are 64 bit and floating point operations for graphical applications are usually done using 32 bit precision. The algorithm is based on propagating already fixed *strong* types through the expression branch of the AST whenever a *strong* type is found, but a complete description is not going be presented for brevity (the code can be found at <https://github.com/Hoshoyo/Light/blob/master/src/type_infer.cpp>).

Every *strong* type found in the type inference step goes through another process of internalization, utilizing a hashing algorithm for types created for the compiler utilizing as a base the FNV hash (FOWLER; VO, 1991). The objective is to create a type table with unique types to later use them for code reflection at runtime.

Type checking of expressions is done alongside type inference. The rules from the chapter 2 are applied and any type mismatch will raise a *Type Error* at this stage. Differently from a *Syntax Error*, this kind of error can be raised more than once. The last part of type checking is to check redeclaration of identifiers within the same scope, which is done by checking the declaration identifier for duplication in the corresponding scope. Other verifications include checking for boolean types in the conditions statements for the commands `if` `while` and `for`, checking if `break` and `continue` commands are inside of loops with compatible depths and type checking return statements with the corresponding procedure return types.

3.4.1 Type Table

Providing type information at runtime allows for reflection, the compiler allows the programmer to manipulate and query data from the type table, kept in the data segment, to construct programs that utilize polymorphic behavior or any other manipulation that is made available by that feature. The ability to generate code through metaprogramming also benefits from this feature, since the textual representation of a type can be trivially generated from the type information available in the type table. A similar program made in *C/C++* would require at least a parser and would still lack important information like type size or alignment.

3.5 Code Generation

The final compilation stage is code generation, in the case of the *Light* compiler, the AST is transformed into *C* code as it is simpler to generate than a more low level machine language such as x64 assembly, although an assembly back end is planned for the future. The current main code generator therefore generates c99 code which at the end calls the *gcc* compiler for both supported platforms (Windows and Linux), making *gcc* a temporary dependency of the compiler along with its linker *ld*.

A second back end was also developed with the intent to run compile time code, for that a small register virtual machine was written with a simple byte code instruction set similar to an x64 architecture. This means that any `#run` directive passes through byte code generation, runs inside the virtual machine and at the end the return value is transformed back into a literal matching the directive expression return type to be finally substituted back into the AST. Though not aimed to be an official back end, this virtual machine is designed to be able to run any *Light* code, maximizing the power of code generation.

To make external calls (calls to the operating system) the virtual machine uses a small part of assembly code which translates its context stack frame to the standard 64 bit *C* calling convention (FOG, 2004), this makes it possible for external linkage at compile time.

For all other nodes besides external procedure call, code generation follows a simple pattern, emit code for each node making note of referenced jumping addresses, in the case of control flow statements, that are later filled in with the appropriate relative or absolute addresses. This technique although not exactly the same as described in the section 6.2 of the book (AHO; SETHI; ULLMAN, 1986), follows a very similar approach to the three-address code, common in many compilers. Register allocation is an important topic in code generation, for this work a very simple algorithm is used, optimization was not a priority in the initial compiler and therefore was not addressed. The current register allocation algorithm for the byte code back end picks the first available register and allocated it, in the case of unavailability, the oldest allocated register is saved into the stack and allocated.

4 COMPARISON TO OTHER PROGRAMMING LANGUAGES

With many languages being proposed each year, it can be argued that great part of this effort is put into ever more abstract and higher level constructs that hide the hardware underneath almost completely. The advance in lower level programming languages, although disproportionately smaller, is noteworthy. Modern languages like *Go*, *Rust*, *D* and others have their own aspects that they consider important in low level language design, each walking different paths. That leaves older languages like *C* and *C++* with the responsibility to adhere or not to modern language philosophy which have shaped them through the decades in arguably good and bad ways, nevertheless they are still heavily used in the industry for high performance computation showing a still needed space for this type of language to evolve. This Chapter presents qualitative comparison between *Light* and alternative languages by means of code examples. It also attempts to point out problems with other languages and reasons why *Light* is a better in certain areas. This Chapter also gives ideas in the overall design path to which development of low level languages should go.

4.1 C

Created in the early 70's along with the Unix operating system, the *C* language was aimed to be a system programming language or sometimes referred to as a higher level assembly language. With a static type system and relatively verbose syntax, *C* stands today as one of the most successful languages ever created, being used to create a plethora of new languages and many other purpose software. Even though the success of C can be attributed to several aspects, an important one is simplicity - when compared to its successor *C++*, *C* is simpler by a great margin. Although a program written in it is sometimes bigger, a relatively experienced *C* programmer can certainly understand it. Some of the arguably more advanced concepts, like pointers, can be a source of a lot of bugs that are certainly unwanted, but it is an example of a necessary construct of the type of language *C* proposes to be. Considering how many years the language has survived and is still widely used, we can infer that the need for a language like *C* is undeniable.

Example 1. *Listing 4.1 presents a code snippet that illustrates the problems with the C* *language that we want to stress. A convoluted syntax contributes to a worse experience for programmers, a simple program like the example shows the lack of syntax clarity of* C *for some language constructs. In the example a function that iterates through pixels of an image pointed by* `unsigned char* image` *- in the commented line 10 the code intent is clear, but because* `image` *is a pointer the compiler cannot calculate the sizes of the array in runtime, resulting in a compile time error. The solution in this case is either casting to a fixed array size at compile time (line 13), or calculating an index and using it directly manipulating memory and using pointer arithmetic for this.*

This example highlights several points that cause friction, leading to syntax confusion that ultimately is not a huge problem but slows down the programming process. A better syntax is ideally consistent and easy to read without having to read carefully to understand what the code is doing. In the *Light* version 4.2 the same code for the type casting to array is in line 9, the consistency with the declaration syntax of an array is direct, whilst an array declaration in *C* is `Type name[size]`, in *Light* is `name : [size]Type` isolating the type and keeping the syntax the same throughout all language constructs.

Listing 4.1 – Example array usage - close to direct memory management

```
1  void modify_image(unsigned char* image, int width, int height) {
2     for(int y = 0; y < height; ++y) {
3        for(int x = 0; x < width; ++x) {
4          // 4 bytes per pixel
5          int index = (y * 4) * width + (x * 4);
6          unsigned char r, g, b;
7          // Calculate rgb values
8          unsigned int color =
9             0xff000000 | (r << 16) | (g << 8) | b;
10
11         // Can't do image[y][x] = color;
12
13         (*(unsigned int (*)[512][512]image)[y][x] = color
14
15         // If width and height are not known at compile time
16         *(unsigned int*)(image + index) = color
17       }
18    }
19 }
```

Listing 4.2 – Light version - array usage

```
modify_image :: (image : ^u8, width : s32, height : s32) {
   for y:s32=0; y < height; y += 1 {
      for x:s32=0; x < width; x += 1 {
         index := (y * 4) * width + (x * 4);
         r, g, b : u8;
         // Calculate rgb values
         color : u32 = 0xff000000 | (r << 16) | (g << 8) | b;

         [ [512][512]u32 ]image[y][x] = color;

         // If width and height are not known at compile time
         *[^u32](image + index) = color;
      }
   }
}
```

Another common example of the same problem is function pointers. While in *C* the declaration name is infixed between parts of the type, making not clear what are the types involved, the same example code written in *Light* is easily read left to right without any ambiguities as is shown in the comparing examples 4.3 and 4.5, where the function `getSum` returns the `sum` function.

Listing 4.3 – C return function pointer

```c
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}

int (*getSum())(int, int) {
    return sum;
}

int main() {
    printf("%d\n", getSum()(2,3));
    return 0;
}
```

Listing 4.4 – C return function pointer - Output

```
$ 5
```

Listing 4.5 – Light return function pointer

```
#import "print.li"

sum :: (a : s32, b : s32) -> s32{
    return a + b;
}

getSum :: () -> (s32, s32) -> s32 {
    return sum;
}

main :: () -> s32 {
    print("%\n", getSum()(2,3));
    return 0;
}
```

Listing 4.6 – Light return function pointer - Output

```
$ 5
```

Another example available at <https://blog.golang.org/gos-declaration-syntax> shows the same problem in both declaration of functions and function pointers in *C* which *Go's* syntax is much more readable. This is also true for *Light* where again, not only types are read from left to right, but they don't differ between different declarations. The Listing 4.7 shows the referred code in *C* and the listing 4.9 the *Light* version.

Listing 4.7 – Unwieldy syntax

```c
#include <stdio.h>

typedef int function_t (int, int);

int sum(int x, int y) {
    return x + y;
}
int sub(int x, int y) {
    return x - y;
}

function_t* transform(int(*f)(int, int), int v) {
    if (f(v, v) > 0) {
        return sum;
    } else {
        return sub;
    }
}

int main() {
    int (*(*fp)(int (*)(int, int), int))(int, int);
    fp = transform;

    printf("%d\n", fp(sum, 3)(4, 5));
    printf("%d\n", fp(sub, 3)(4, 5));
    return 0;
}
```

Listing 4.8 – Unwieldy syntax - Output

```
$ 9
$ -1
```

Listing 4.9 – Unwieldy syntax - Light version

```
#import "print.li"

sum :: (x : s32, y : s32) -> s32 {
    return x + y;
}
sub :: (x : s32, y : s32) -> s32 {
    return x - y;
}

transform :: (f : (s32, s32) -> s32, v : s32) {
   if f(v, v) > 0 {
      return sum;
   } else {
      return sub;
   }
}

main :: () -> s32 {
   fp := transform;

   print("%\n", fp(sum, 3)(4, 5));
   print("%\n", fp(sub, 3)(4, 5));

   return 0;
}
```

To illustrate this, a procedure declaration in Light follows the pattern

```
name :: (arg1 : s32, arg2 : string) -> s32
```

where each argument inside parentheses is identical to a variable declaration and the return type comes after the -> token, while a type declaration `(s32, string)-> s32` of this function type follows the same pattern, omitting the names and the `::` token - which means constant declaration. If it is not apparent in that simple example, the same example given in the Chapter 5.12 Complicated Declarations of the book *The C programming language* (KERNIGHAN, 1988), is read left to right in a simpler manner in Light.

C	*Light*	*Description*
`int *f();`	`f :: ()-> ^int;`	f: function returning pointer to int
`int (*daytab)[13];`	`daytab : [13]^int;`	daytab: array[13] of pointer to int
`int (*pf)();`	`f : ^()-> int;`	fp: pointer to function returning int
`char (*(*x[3])())[5];`	`x : [3]^()-> ^[5]char`	x: array[3] of pointer to function returning pointer to array[5] of char

4.2 C++

Created in 1979 by Bjarne Stroustrup as a ”C with classes”, *C++* introduced the object oriented paradigm while maintaining direct compatibility with *C's* procedural style and its standard library. *C++'s* feature set is one of the biggest and most complex feature sets of lower level programming languages whilst tooling and support are also one of the biggest and most mature. The consequences for this large feature set are lack of consistency in general, making the language prone to errors which can be harder to avoid as a project grows forcing projects to have guidelines or even to prohibit some of the language features completely from being used. Louis Brandy, developer for facebook, talks about several problems that can occur to large code bases due to this lack of consistency and overload of features in his talk at CppCon 2017 (BRANDY, 2017).

Many of the features currently in *C++* were designed and added after the initial language definition, an example is the runtime type information or RTTI, although available in *C++*, it is very limited, as the example shows, only the name and a hash of a given structure or class can be retrieved, also types are comparable like shown in line 23 of the example 4.10. In this example, a structure `Entity` can have its name accessed at compile time, but the name of fields or type information are not provided.

Listing 4.10 – Limited runtime type information

```
#include <iostream>
#include <typeinfo>

struct Entity {
    char name[32];
    int age;
};

int main(int argc, char** argv) {
    const std::type_info& info = typeid(Entity);

    Entity e = {
        "entityName",
        20
    };

    std::cout << typeid(e).name() << std::endl;
    std::cout << typeid(e).hash_code() << std::endl;

    Entity f;
    Entity g;

    if(typeid(f) == typeid(g)) {
        std::cout << "Equal types" << std::endl;
    } else {
        std::cout << "Not equal types" << std::endl;
    }

    return 0;
}
```

For *C++11*, `constexpr` was added as a way to run code at compile time. This may be considered enough for simple constant functions, but is limited as no external functions can be called as shown in the example 4.11 where a simple hashing function (line 4) is compiled successfully whilst a compilation error is thrown when trying to call a library function `printf` (line 22).

Listing 4.11 – Limited compilation time execution

```
1  typedef unsigned long long u64;
2  
3  // Fowler-Noll-Vo hash function
4  constexpr u64 fnv1_hash(char* s, u64 length) {
5     u64 hash = 14695981039346656037;
6     u64 fnv_prime = 1099511628211;
7  
8     for(u64 i = 0; i < length; ++i) {
9        hash = hash * fnv_prime;
10       hash = hash ^ s[i];
11    }
12 
13    return hash;
14 }
15 
16 int main(int argc, char** argv) {
17    printf("%llu", fnv1_hash("Hello", sizeof("Hello") - 1));
18    return 0;
19 }
20 
21 // Compilation error
22 constexpr void print(char* str) {
23    printf("%s", str);
24 }
```

Template meta-programming started as a feature to aid programmers in generic programming and was not designed for general purpose. Quickly after the realisation that templates are turing complete in *C++*, illustrated in the article by Todd L. Veldhuizen (VELDHUIZEN, 2003), *C++* programmers started using as a way to run arbitrary code at compile time, in the example 4.12 a factorial function is defined using templates.

Listing 4.12 – Template meta-programming

```
template <int N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0> {
    enum { value = 1 };
};
```

4.3 Go

Go is a language created by Google with the simplicity design philosophy in mind, the main designers of the language are Robert Pike and Ken Thompson, the latter also a creator of the *C* language. *Go* however, was not designed to be a system's language, offering memory management through garbage collection and a sizable runtime support, even though a statically typed compiled language, its priority is productivity above control and speed.

Robert Pike in his talk ”Simplicity is Complicated” in 2015 (PIKE, 2015) explaining the success of *Go*, says that to have simplicity *Go* has hidden a good amount of complexity, which *Light's* design tries to avoid even though it might hinder simplicity in the language's front end to get simplicity in the back end in order to make the back end also visible and understandable by the programmer, and this way offering a large amount of control over the code.

Opting also to have limited meta-programming capabilities, *Go* feels like a more friendly and solid *C* while focusing efforts in features to help concurrent and distributed programming. As it follows a very similar design principle as *Light's*, *Go* also inspired some decisions in the creation of *Light*, mainly syntax, type inference and out of order top level declarations.

4.4 Rust

Rust proposed to be an alternative for system's programming by avoiding the need for a garbage collector with clever use of ownership and borrowing semantics making memory allocation errors less of a concern to the programmer. This approach to safety may encourage a different approach to memory management but also locks it artificially as it can be circumvented by creating custom memory allocators, which is common in lower level programming. *Light* addresses the safety issue not by adding features to the language, but making the language friendly to debugging and troubleshooting by providing good meta-programming support for writing helper tools, customizable code checking and relying on visualization tools to catch memory errors.

Going in a completely different approach as the base language, *Rust* also provides meta-programming support in a form of macros which heavily make use of pattern matching and introduce several new syntactical features. Similar to *C++'s* template features, *Rust* introduces new concepts which don't match the base language and therefore can arguably be considered new languages within the originals and therefore resulting in the growth of the language complexity.

4.5 D

Very similar to *C++*, *D* retains a heavy object orientation paradigm along with most of the features that characterize *C++*, mainly RAII, template meta-programming and exception handling. As it is still a low level programming language, *D* supports important features like inline assembly for x86 and x64 maintaining hardware as a language concern instead of abstracting it completely. Although *D's* design didn't allow for contentious features like multiple inheritance and direct *C* compatibility for simplicity of the language, its complexity is still comparable to *C++'s*.

5 EXPERIMENTS AND VALIDATION

One of the main objectives of the Light compiler is to provide with the maximum compilation speed possible, and this principle affected many compiler design decisions. Table 5.1 shows examples of compilation of programs with different amounts in lines of code and compared the *Light* compiler complete run with the *gcc* compiler running in the same machine (no optimizations are turned on). The results are an average of ten consecutive compiler executions. Although the *Light* compiler currently relies on generating *C* code, we believe a corresponding Assembly back end would have similar generation time. This is encouraging evidence regarding the efficiency of the *Light* compiler. We believe that optimizing its code, which is currently single threaded, to a multi threaded version would improve compilation even more, since compilation stages such as the parser and lexer could be independently processed for every source file.

Table 5.1: Compile time - i7-2600 3.40 GHz

Lines of code	Light Only (ms)	gcc (ms)	Light with gcc backend (ms)
4651	24.72	492.59	517.31
507	2.98	155.38	158.36
334	0.91	129.8	130.71
30	0.76	108.62	109.38

In order to test code running time, a small benchmark was created to compare some of the most popular languages nowadays. The code can be found at <https://github.com/Hoshoyo/LanguagesBenchmark>. In this example the famous *Mandelbrot set* (MANDELBROT et al., 2004) image with dimensions 800 by 800 pixels, was calculated using 256 iterations to check for escape, meaning each pixel iterates 256 times at the worst scenario (if it is not in the Mandelbrot set), the result miliseconds are a mean of six consecutive runs of the same program. Although optimizations for the initial version of the compiler, as already mentioned, were defered to future work, the results were obtained from the current *C* back end. In Table 5.2 we can see that *Light* language ranks among the fastest runtimes - Javascript ranking is optimizing for usage of multiple cores while the other versions are all single threaded. The results represent an average of ten consecutive executions.

Table 5.2: Mandelbrot benchmark - i7-2600 3.40 GHz

Language	Elapsed time (ms)	Standard Deviation (ms)
C++/g++	362.10	1,31
Light/g++	368.50	1,35
Javascript	387.17	5,01
Java	603.45	4,44
Matlab	6511.98	79,92
PHP	67042.02	1231,84
Python	218594.75	2487,09

Several other examples are available in the compiler public repository, which include utility libraries, common data structures, language feature demos and more. Among the most complex examples are a small graphical engine with working OpenGL bindings, a simple server and an implementation of the fast fourier transform.

6 FUTURE WORK

The Light language is still under development. At this stage we have the complete language core, runtime type information, type inference and an incomplete implementation for compile time code execution. However, there are many language features which were planned but are not available in the current state of the language. We now revise the most important planned additions.

The first step in completing the language is to implement key defining meta-programming features described in this work. Compile time execution of code currently does not have a context from which to run, this would be solved by implementing a dependency system where the compiler can use only declarations within the scope of the `#run` directive to execute it. Still regarding meta-programming, code modification is planned to be a messaging system where the programmer can, at compile time, modify the AST to perform checks or generate arbitrary code to perform a task at specific points in the code. Other unimplemented meta-programming features already described in this work are: static assertion, static *if/else* statements, the `#compile` and the `#export` directives.

An important feature for any language is its library modules support. Although not yet defined, an import dependency system similar to *Python's* is considered a good alternative to provide support for libraries with different defined namespaces. Another alternative, for example, is a packaging system similiar to what the *Go* language implements, where source files within a directory constitute a package, which defines its own namespace. Expose compiler bindings to be used as a library is also an important feature that allows for tools to use the compiler as a library to, for example, provide syntax highlight to an editor by using the compiler parser in a file. This also would allow for generation of debugging information to debuggers such as gdb (STALLMAN, 1988).

An alternative to *C++'s* RAII way of resource managing, also used by the *Go* language is the defer statement. This would allow an easy and explicit way to execute code at the end of scope blocks and procedure returns. A simple example is the freeing of memory or closing a file handle using the defer statement, this makes managing resources in the same scope more clear.

The current version supports only a *C* back end. An initial goal for a more definitive back end is to provide a simple x64 Assembly back end without optimizations. This would eliminate the dependency on the gcc compiler leaving only the dependency for the linker. To eliminate this dependency, the next step would be to generate PE/COFF (MI-

CROSOFT, 2018) and ELF64 (LABORATORIES,) executable files for Windows and Linux respectively. Although this eliminates the gcc dependency, to link with *C* libraries, a linker would still have to be written. The goal of this back end is to have an efficient *Debug* build. To provide good optimization, we plan to provide an option for an LLVM IR (LLVM. . . ,) back end, which would utilize the latest advances in compiler optimizations to generate the most efficient runtime code possible for *Release* builds.

Error messages are also considered very important to have good productivity. We plan to improve error messages for type mismatch to provide better description of the context in which the error occured. A code path analyzer to report missing return statements is also planned with the goal to maximize static type checking.

Runtime type information is planned to be used along variadic argument procedures to provide type information to variadic functions. This eliminates the need for unsafe markers in functions such as `printf`, since type information can be accessed by the function at runtime. Compatibility with the *C* calling convention in that regard is still undefined.

Finally, we plan to write a standard library consistent with the main goals of the language and compatible with modern technologies.

7 CONCLUSION

This work presented the design and implementation of Light, a low level programming language with support for meta-programming.

The Light language is based on a simple, imperative core language with clear syntax. This core language, although low level, allows for modern features such as type inference and literals for structure types. A complete language documentation was not yet provided, since the language current state is still changing. But the provided implementation supports all core language constructs making possible to construct working complex example programs.

One distinct feature of Light is that it provides compiler support for meta-programming techniques. Compiler directives can invoke the compiler to execute arbitrary code during compilation. This choice makes possible to use meta-programming as a tool for implementing tasks usually performed by pre-processors and scripts in C/C++.

Besides the language design, a compiler for Light was developed in C++. Although the current compiler generates C code, relying on GCC for code generation, a direct Assembly backend is planned for the language. Meta-programming, on the other hand, in particular, relies on compilation to bytecode and bytecode interpretation. Early experimental data indicates that the compiler is lightweight and provide fast compilation.

Upon the completion of all its most interesting features, we expect Light to become a viable, modern alternative to the currently available low level languages for systems programming.

REFERENCES

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers principles, techniques, and tools**. Reading, MA: Addison-Wesley, 1986.

BENNETT, J. P. **Introduction to Compiling Techniques: A First Course Using ANSI C, LEX and YACC**. 2nd. ed. New York, NY, USA: McGraw-Hill, Inc., 1996. ISBN 007709221X.

BLOW, J. **Ideas about a new programming language for games.** YouTube, 2014. Available from Internet: <https://www.youtube.com/watch?v=TH9VCN6UkyQ&list=PLmV5I2fxaiCKfxMBrNsU1kgKJXD3PkyxO>.

BRANDY, L. **Curiously Recurring C++ Bugs at Facebook - CppCon**. 2017. Available from Internet: <https://www.youtube.com/watch?v=lkgszkPnV8g>.

DEVELOPERS, G. **Go language documentation**. 2018. Available from Internet: <https://golang.org/doc/>.

DEVELOPERS, T. R. P. **Rust Documentation**. 2011. Available from Internet: <https://doc.rust-lang.org/>.

FOG, A. **Calling conventions for different C++ compilers and operating systems**. 2004. Available from Internet: <https://www.agner.org/optimize/calling_conventions.pdf>.

FOUNDATION, D. L. **D Language Reference**. 1999. Available from Internet: <https://dlang.org/>.

FOWLER, L. C. N. G.; VO, K.-P. **FNV Hash**. 1991. Available from Internet: <https://tools.ietf.org/html/draft-eastlake-fnv-15#section-6>.

GRUNE, D. et al. **Modern Compiler Design**. 2nd. ed. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 1461446988, 9781461446989.

IEEE Standard for Floating-Point Arithmetic. **IEEE Std 754-2008**, p. 1–70, Aug 2008.

INTEL. **Intel® 64 and IA-32 Architectures Software Developer Manuals**. [S.l.], 2016.

JOHNSON, S. C. et al. **Yacc: Yet another compiler-compiler**. [S.l.]: Bell Laboratories Murray Hill, NJ, 1975.

JUNG, R. et al. Rustbelt: Securing the foundations of the rust programming language. **Proc. ACM Program. Lang.**, ACM, New York, NY, USA, v. 2, n. POPL, p. 66:1–66:34, dec. 2017. ISSN 2475-1421. Available from Internet: <http://doi.acm.org/10.1145/3158154>.

KERNIGHAN, B. W. **The C Programming Language**. 2nd. ed. [S.l.]: Prentice Hall Professional Technical Reference, 1988. ISBN 0131103709.

KNUTH, D. E. **The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-89683-4.

LABORATORIES, U. S. **Executable and Linking Format (ELF) - Manual**. Available from Internet: <http://man7.org/linux/man-pages/man5/elf.5.html>.

LLVM IR Documentation. Available from Internet: <https://llvm.org/docs/LangRef.html>.

MANDELBROT, B. et al. **Fractals and Chaos: The Mandelbrot Set and Beyond**. Springer, 2004. (Selecta (Springer)). ISBN 9780387201580. Available from Internet: <https://books.google.com.br/books?id=As2xeIwS6OgC>.

MICROSOFT. **PE-coff Documentation**. 2018. Available from Internet: <https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format>.

MUCHNICK, S. S. **Advanced Compiler Design and Implementation**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55860-320-4.

PIERCE, B. C. **Types and Programming Languages**. 1st. ed. [S.l.]: The MIT Press, 2002. ISBN 0262162091, 9780262162098.

PIKE, R. **dotGo - Simplicity is Complicated**. 2015. Available from Internet: <https://www.youtube.com/watch?v=rFejpH_tAHM>.

STALLMAN, R. **GDB manual: the GNU source-level debugger**. 2nd, GDB version 2.5. ed. Cambridge? Mass., 1988. ii + 63 p.

STRACHEY, C. Fundamental concepts in programming languages. **Higher-Order and Symbolic Computation**, v. 13, n. 1, p. 11–49, Apr 2000. ISSN 1573-0557. Available from Internet: <https://doi.org/10.1023/A:1010000313106>.

STROUSTRUP, B. **An Overview of the C++ Programming Language**. 1999.

VELDHUIZEN, T. L. **C++ Templates are Turing Complete**. [S.l.], 2003.

WESTRUP, E.; PETTERSSON, F. **Using the Go Programming Language in Practice**. 2014. (Department of Computer Science, Faculty of Engineering LTH). Student Paper.

APPENDIX A — GRAMMAR

command := **{ }**
 | **{** *helper command list* **}**
 | *command variable assignment*
 | *command if*
 | *command for*
 | *command while*
 | *command break*
 | *command continue*
 | *command return*

comma separated commands := *command*
 | *command* **,** *comma separated commands*

helper command list := *command*
 | *commandcommand helper list*

operator assignment := **=** | **+=**
 | **-=** | ***=**
 | **/=** | **%=**
 | **<<=** | **>>=**
 | **^=** | **&=**
 | **|=**

command variable assignment := *lvalue expression* *operator assignment* *expression* **;**

command if := **if** *expression* *command*
 | **if** *expression* *command* **else** *command*

command for := **for** *comma separated commands* **;** *expression* **;** *comma separated commands* *command*

command while := **while** *expression* *command*

command break := **break** *int literal* **;**
 | **break** **;**

command continue := **continue** **;**

command return := **return** *expression***;**
 | **return** **;**

top level := *declaration list*

declaration list := *declaration*

| *declaration list*

declaration := *declaration procedure*

| *declaration variable*

| *declaration struct*

| *declaration constant*

| *declaration union*

| *declaration enum*

declaration variablelist := *declaration variable*

| *declaration variable* **,** *declaration variable list*

declaration arguments list := *declaration variable*

| *declaration variable* **;** *declaration arguments list*

declaration constant list := *declaration constant*

| *identifier*

| *declaration constant* **,** *declaration constant list*

| *identifier* **,** *declaration constant list*

declaration procedure := *identifier* **::** **(** **)** **->** *type* **{** *command list* **}**

| *identifier* **::** **(** *declaration variable list* **)** **->** *type* **{** *command list* **}**

declaration variable := *identifier* **:** *type*

| *identifier* **:** *type* **=** *literal*

declaration struct := *identifier* **::** *struct* **{** *declaration arguments list* **}**

declaration struct := *identifier* **::** *union* **{** *declaration arguments list* **}**

| *identifier* **:** *type* **:** *union* **{** *declaration arguments list* **}**

declaration constant := *identifier* **::** *literal*

| *identifier* **::** *constant*

| *identifier* **:** *type* **:** *literal*

| *identifier* **:** *type* **:** *constant*

expression := `(` *expression* `)`
| *expression binary*
| *expression unary*
| *expression literal*
| *expression variable*

operator unary prefixed := `-` | `+`
| `*` | `&`
| `~` | `!`
| `[` *type* `]`

operator binary := `-` | `+`
| `*` | `/`
| `%` | `&`
| `|` | `^`
| `&&` | `||`
| `<<` | `>>`
| `<` | `>`
| `<=` | `>=`
| `==` | `!=`
| `.`

expression unary := *operator unary prefixed* *expression*

expression binary := *expression* *operator binary* *expression*
| *expression* `[` *expression* `]`
| *expression* `(` `)`
| *expression* `(` *expression list* `)`

expression variable := *identifier*

expression literal := *literal int*
| *literal float*
| *literal bool*
| *literal struct*
| *literal array*

literal int := `[0-9]+`
| `0x([0-9]|[a-f]|[A-F])+`
| `0b(0|1)+`
literal float := `[0-9]+.[0-9]+`
literal bool := `true`
| `false`
literal pointer := `null`
literal string := `\" (\\.|[^"\\] )* \"`
identifer := `([a-z]|[A-Z]|_)([a-z]|[A-Z]|_|[0-9])*`
literal struct := **struct** *identifier* **{** **}**
| **struct** *identifier* **{** *literal list* **}**
literal list := *literal*
| *literal* **,** *literal list*
literal array := **array** **{** **}**
| **array** **{** *literal list* **}**
expression list := *expression*
| *expression* **,** *expression list*
expression directive := **#** **sizeof** *type*
| **#** **typeof** *expression*
| **#** **run** *expression*
| **#** **assert** *expression*
| **#** **import** **(** *literal string* **)**

type	:=	*type primitive*
	|	*type ptr*
	|	*type struct*
	|	*type array*
	|	*type function*
type primitive	:=	**s8** | **s16** | **s32**
	|	**s64** | **u8** | **u16**
	|	**u32** | **u64** | **r32**
	|	**r64** | **bool** | **void**
type ptr	:=	**^** *type*
type struct	:=	*identifier*
type array	:=	**[** *int literal* **]** *type*
	|	**[** *constant name* **]** *type*
type list	:=	*type*
	|	*type* **,** *type list*
type function	:=	**()** **->** *type*
	|	**(** *type list* **)** **->** *type*

APPENDIX B — ABSTRACT SYNTAX TREE

Listing B.1 – Light's AST

```
// ----------------------------------------
// ------------ Declarations --------------
// ----------------------------------------

struct Ast_Decl_Procedure {
        Token*         name;
        Ast**          arguments;             // DECL_VARIABLE
        Ast*           body;                  // COMMAND_BLOCK
        Type_Instance* type_return;
        Type_Instance* type_procedure;
        Scope*         arguments_scope;

        Site   site;

        u32    flags;
        s32    arguments_count;

        u64*   proc_runtime_address;

        Token* extern_library_name;
};

struct Ast_Decl_Variable {
        Token*         name;
        Ast*           assignment;            // EXPRESSION
        Type_Instance* variable_type;

        Site site;

        u32 flags;
        s32 size_bytes;
        s32 alignment;
        u32 temporary_register;
        s32 stack_offset;
        s32 field_index;
};
```

```

struct Ast_Decl_Struct {
        Token*          name;
        Ast**           fields;                    // DECL_VARIABLE
        Type_Instance* type_info;
        Scope*          struct_scope;

        Site site;

        u32 flags;
        s32 fields_count;
        s32 alignment;
        s64 size_bytes;
};
struct Ast_Decl_Union {
        Token* name;
        Ast** fields;
        Type_Instance* type_info;
        Scope* union_scope;

        Site site;

        u32 flags;
        s32 fields_count;
        s32 alignment;
        s64 size_bytes;
};

struct Ast_Decl_Enum {
        Token*          name;
        Ast**           fields;                    // DECL_CONSTANT
        Type_Instance* type_hint;
        Scope*          enum_scope;

        Site site;

        u32 flags;
        s32 fields_count;
};
struct Ast_Decl_Constant {
        Token*          name;
```

```
        Ast*           value;           // LITERAL | CONSTANT
        Type_Instance* type_info;

        Site site;

        u32 flags;
};

struct Ast_Decl_Typedef {
        Token*         name;
        Type_Instance* type;

        Site site;
};

// ----------------------------------------
// -------------- Commands ----------------
// ----------------------------------------

struct Ast_Comm_Block {
        Ast**   commands;        // COMMANDS
        Scope* block_scope;
        Ast*    creator;
        s32     command_count;
};
struct Ast_Comm_VariableAssign {
        Ast*    lvalue;   // EXPRESSION
        Ast*    rvalue;   // EXPRESSION
};
struct Ast_Comm_If {
        Ast* condition;          // EXPRESSION (boolean)
        Ast* body_true;          // COMMAND
        Ast* body_false;         // COMMAND
};
struct Ast_Comm_For {
        Ast* condition;          // EXPRESSION (boolean)
        Ast* body;                       // COMMAND
        s64  id;
};
struct Ast_Comm_Break {
```

```
        Ast*   level;                    // INT LITERAL [0, MAX_INT]
        Token* token_break;
};
struct Ast_Comm_Continue {
        Token* token_continue;
};
struct Ast_Comm_Return {
        Ast*   expression;       // EXPRESSION
        Token* token_return;
};

// ----------------------------------------
// ------------- Expressions --------------
// ----------------------------------------

struct Ast_Expr_Binary {
        Ast* left;
        Ast* right;
        Token*          token_op;
        Operator_Binary op;
};

const u32 UNARY_EXPR_FLAG_PREFIXED  = FLAG(0);
const u32 UNARY_EXPR_FLAG_POSTFIXED = FLAG(1);
struct Ast_Expr_Unary {
        Ast*           operand;
        Token*         token_op;
        Operator_Unary op;
        Type_Instance* type_to_cast;
        u32            flags;
};

struct Ast_Expr_Literal {
        Token*       token;
        Literal_Type type;
        u32 flags;
        union {
                u64 value_u64;
                s64 value_s64;

```

```
                r32 value_r32;
                r64 value_r64;

                bool value_bool;

                Ast**  struct_exprs;
                struct {
                        Ast** array_exprs;
                        Type_Instance* array_strong_type;
                };
        };
};

struct Ast_Expr_Variable {
        Token* name;
        Ast*   decl;
};

struct Ast_Expr_ProcCall {
        Ast*   caller;
        Ast**  args;            // EXPRESSIONS
        s32    args_count;
};

struct Ast_Data {
        Data_Type type;
        u8*       data;
        s64       length_bytes;
        Token*    location;
        Type_Instance* data_type;
        s32       id;
};

struct Ast_Expr_Directive {
        Expr_Directive_Type type;
        Token* token;
        union {
                Ast*           expr;
                Type_Instance* type_expr;
        };
};
```

```

struct Ast {
        Ast_NodeType    node_type;
        Type_Instance*  type_return;
        Scope*          scope;

        s64 infer_queue_index;
        u32 flags;

        union {
                Ast_Decl_Procedure      decl_procedure;
                Ast_Decl_Variable       decl_variable;
                Ast_Decl_Struct         decl_struct;
                Ast_Decl_Union                  decl_union;
                Ast_Decl_Enum           decl_enum;
                Ast_Decl_Constant       decl_constant;
                Ast_Decl_Typedef                decl_typedef;

                Ast_Comm_Block          comm_block;
                Ast_Comm_VariableAssign comm_var_assign;
                Ast_Comm_If             comm_if;
                Ast_Comm_For            comm_for;
                Ast_Comm_Break          comm_break;
                Ast_Comm_Continue               comm_continue;
                Ast_Comm_Return         comm_return;

                Ast_Expr_Binary         expr_binary;
                Ast_Expr_Unary          expr_unary;
                Ast_Expr_Literal        expr_literal;
                Ast_Expr_Variable       expr_variable;
                Ast_Expr_ProcCall       expr_proc_call;

                Ast_Expr_Directive      expr_directive;

                Ast_Data                data_global;
        };

        s32 unique_id;
};
```